ChatGPT

IAMF-Zero Security Blueprint

This document provides a comprehensive security blueprint for the IAMF-Zero implementation, focusing on protecting data flows and preventing unauthorized access across its components (Pharos, Telos, Osiris, Phoenix, etc.). It outlines common threats (data interception, privilege escalation, unauthorized role execution), and prescribes technical countermeasures (TLS/mTLS, encrypted messaging, signed JWTs, API gateway policies, RBAC/ABAC, logging) to mitigate them. All guidance aligns with Zero Trust principles – **"never trust, always verify"** [1] [2] – treating every service call as untrusted until authenticated and authorized.

Threat Model and Risk Scenarios

- **Data Interception (Eavesdropping):** Attackers may sniff network traffic or message queues to steal or manipulate sensitive data (AI outputs, policy decisions, user data). Without encryption, man-in-the-middle (MITM) attacks can reveal internal messages [3].
- **Privilege Escalation:** A compromised service or stolen credentials could be used to gain higher privileges. For example, if an attacker obtains a Pharos service token, they might perform unauthorized output filtering or gain access to Telos policies. Strict access controls and least-privilege defaults are needed to prevent one module or actor from climbing privilege levels.
- **Unauthorized Role Execution:** An adversary (or buggy component) might try to invoke a role or service it shouldn't. For example, forging a request as the Ethics Gatekeeper (Osiris) to override policy, or invoking Phoenix lifecycle actions without permission. Ensuring strong service identities and policy checks prevents modules from impersonating others.
- **Other Threats:** Injection attacks on APIs, replay of stale messages, and insider threats are also concerns. All input should be validated, and message replay should be guarded by nonces or timestamps. Comprehensive logging (see below) helps detect anomalies.

Secure Communication Channels

- **Transport Encryption (TLS/mTLS):** All network links between IAMF services must use TLS. Mutual TLS (mTLS) is strongly recommended between services: both client and server present certificates, so each party cryptographically verifies the other's identity [4]. TLS guarantees confidentiality and integrity of data in flight: "TLS is an encryption technology that prevents unintended eavesdropping on messages" [3]. In practice, enable HTTPS (HTTP/TLS) on all REST/gRPC endpoints and configure service meshes or sidecars to enforce mTLS automatically [4] [5].

- **Encrypted Message Queues:** If using asynchronous messaging (e.g. Kafka, RabbitMQ), enable TLS for broker connections and client libraries. For instance, Apache Kafka supports TLS+SASL (JAAS) so that brokers encrypt topics in transit and authenticate producers/consumers [3]. This prevents attackers from reading or injecting into queues. Message brokers should also implement access controls (e.g. ACLs, JAAS) so that only authorized services can publish or consume particular topics.

- **Segmentation and Isolation:** Apply network segmentation or Kubernetes NetworkPolicies to isolate IAMF modules. Treat every network hop as untrusted. Even internal service-to-service calls

must be encrypted [1] [2]. Use zero-trust network functions (identity-aware proxies or service mesh) to enforce these links.

- **Infrastructure Hardening:** Disable insecure protocols. Rotate TLS certificates regularly and use strong ciphers. Consider hardware security modules (HSMs) or cloud KMS for key management. All data repositories (databases, disks) should also use encryption-at-rest to protect logs and databases if disks are compromised.

Authentication and Access Control

- **Signed JWTs & Identity Tokens:** Adopt a central identity provider (IdP) using OAuth2/OpenID Connect. Each IAMF service or API call should carry a signed JWT (JSON Web Token) issued by the IdP [6]. The JWT's payload should include the service's identity (e.g. `sub=pharos`) and roles/scopes. For example, Pharos might send a token with `roles=["OutputValidator"]`. Recipients validate the JWT signature and ensure it is not expired [6]. This prevents spoofing: "A JWT has a data payload signed by a trusted party to prevent spoofing" [7]. Use asymmetric signing (RS256) so each microservice can verify tokens without sharing a symmetric key.

- **Role-Based and Attribute-Based Access Control:** Implement RBAC/ABAC inside each component and at the gateway. In RBAC, each service or user is granted permissions according to a predefined role [8]. For example, only the Osiris (Ethics Gatekeeper) role can execute hard rule checks. ABAC can complement RBAC for finer-grained rules using attributes (time of day, location, sensitivity level) [9]. For instance, an access policy might allow Telos updates only during business hours or for certain environments. In practice, encode user/service attributes in JWT claims and enforce ABAC rules via a policy engine.

- **Policy Engine (OPA):** Use a dedicated policy decision point (PDP) such as Open Policy Agent. Services act as Policy Enforcement Points (PEPs) that query the PDP for access decisions. This decouples authorization logic: policies (written in Rego or ALFA/XACML) can express complex ABAC rules. The PDP returns "allow/deny" for each request based on attributes. Ensure that every decision is logged [10] (see Logging below).

- **API Gateway Policies:** Configure the API gateway (or service mesh) to enforce authentication and coarse authorization before requests reach any service. For example, the gateway can verify JWTs and reject any call missing a valid token. It can map JWT scopes to internal routing, and enforce global RBAC rules (e.g. only certain roles can invoke Telos endpoints). The gateway may also integrate with OPA or a similar policy evaluator to centralize authorization rules.

- **Least Privilege & Segregation:** Assign each service a unique identity (certificate or service account) and only grant it the minimum permissions it needs. Do not share credentials between services. For example, Pharos's service account should never be allowed to modify Telos's data. Use Kubernetes RBAC or cloud IAM policies to restrict Kubernetes API access or cloud resources per service account. Regularly audit these permissions.

API Gateway & Centralized Enforcement

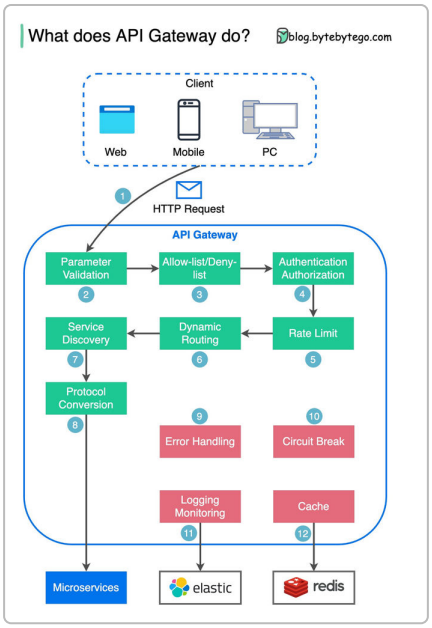

Figure: An API gateway provides a unified front-door for all microservice requests. It performs parameter validation, allow-list/deny-list checks, authentication/authorization (e.g. JWT or mTLS), rate limiting, routing, protocol conversion, error handling, and integrated logging/caching.

Use an API gateway or ingress controller to centralize security controls [11]. The gateway should:

- **Validate Credentials:** Check incoming JWTs or mTLS client certificates. Only requests with valid tokens/certificates (issued by the trusted CA/IdP) should be forwarded [6] [4].
- **Enforce Access Policies:** Apply global rules (allow/deny lists, role-scoped ACLs, and ABAC policies). For instance, restrict which services or endpoints a caller can reach, and check token scopes against endpoint requirements.
- **Rate Limiting and Throttling:** Limit traffic per client or per role to prevent abuse. Rate limits mitigate denial-of-service attempts.
- **Protocol Translation & Routing:** Convert protocols if needed (e.g. REST to gRPC) and dynamically route to the correct backend service.
- **Logging & Monitoring:** Log all requests (with correlation IDs) for auditing and alerting. The gateway can enrich logs with client identity and decision outcomes.

By enforcing authentication and authorization at the gateway layer, the internal services can assume requests arriving downstream have already been pre-checked. This also simplifies how Pharos, Telos, Osiris, etc. trust incoming calls – they only trust connections that passed gateway and mTLS validation [4].

Logging and Auditing

Maintain detailed, tamper-evident logs for transparency and compliance. Recommended practices:

- **Comprehensive Audit Trails:** Record every significant security-related event: authentication attempts, token issuances/validations, authorization decisions, data access (reads/writes), configuration changes, and error events [12] [10]. For example, log when Pharos requests policy

data from Telos, or when Osiris rejects an output. Each audit record should include who/what performed the action, what the action was, which resource was affected, and the outcome (allow/deny).

- **Centralized Logging:** Use a log aggregation system (ELK/EFK stack, Splunk, or cloud logging) to collect logs from all IAMF modules. Standardize log formats (JSON) and include contextual metadata (timestamps, trace IDs, user/service IDs) so events can be correlated end-to-end.

- **Immutable Storage:** For compliance, write audit logs to a write-once medium or an append-only ledger. Protect logs with strict access controls so attackers cannot alter or delete them. Consider digital signatures or Merkle trees for log chains.

- **Real-Time Monitoring and Alerts:** Continuously monitor logs for anomalies (e.g. repeated failed auth attempts, privilege escalations). Generate alerts on suspicious patterns. This complements the Zero Trust ideal of continuous verification – even authorized sessions should be reevaluated if behavior changes.

- **Regular Audits:** Periodically review access logs and policy logs. Use them to demonstrate compliance (e.g. who accessed which data when). Logging details ensures "security teams can check for suspicious requests" [10] and provides forensic data if an incident occurs.

Example: Secure Pharos→Osiris Communication

To illustrate, consider Pharos sending an AI output to Osiris for hard-rule evaluation. Both services authenticate and encrypt the channel. For instance, using Python pseudocode:

```
import ssl, requests, jwt

# Generate a signed JWT for Pharos's identity (issuer=IAMF, subject=pharos, role=OutputValidator)
private_key = open("pharos_private.pem").read()
token = jwt.encode(
    {"iss": "IAMF-Zero", "sub": "pharos", "roles": ["OutputValidator"], "exp": 1685700000},
    private_key, algorithm="RS256"
)

# Configure mutual TLS with Pharos client cert and key
session = requests.Session()
session.verify = "ca_bundle.pem"                # trust CA that signed Osiris cert
session.cert = ("pharos_client.crt", "pharos_client.key")

# Send HTTPS POST to Osiris API with JWT in the Authorization header
response = session.post(
    "https://osiris.iamf.internal/evaluate",
    headers={"Authorization": f"Bearer {token}"},
    json={"ai_output": model_output}
)

if response.status_code == 200:
    print("Osiris accepted the output.")
```

```
else:
    print("Denied:", response.text)
```

In this example: - **mTLS**: The `session` uses a client certificate (`pharos_client.crt/key`) signed by the trusted CA. Osiris will only accept connections from clients with valid certificates [4]. This authenticates Pharos's host identity at the TLS layer.
- **JWT**: Pharos includes a signed JWT in the header. Osiris's application verifies the token's signature (preventing spoofing) and checks the claims (e.g. `roles=["OutputValidator"]`) to ensure Pharos is allowed to invoke this endpoint [6] [7].
- **Least Privilege**: The JWT scope only permits what Pharos needs. Even if an attacker steals `pharos_client.key`, they cannot use it to perform other roles or access unrelated data because the token's roles are fixed and other endpoints will reject it.
- **Logging**: Osiris logs the JWT subject, source IP, and policy decision for the request. If Pharos's token were invalid or revoked, Osiris would log a warning and return an error without executing any rule.

This pattern – mTLS + signed tokens on each call – prevents data interception and impersonation, ensuring Pharos-to-Osiris communication is confidential, authenticated, and auditable [4] [7].

Zero Trust Alignment

IAMF-Zero's security must fully embody **Zero Trust**: never assume trust for any request or service. In practice, this means:

- **Verify Every Request:** As NIST recommends, "never trust, always verify" – every access request is checked dynamically against current credentials and policies [1]. Even internal traffic between Pharos, Telos, etc. is authenticated and authorized. There are no trusted "internal networks" or implicit permissions [1] [2].

- **Least Privilege:** Services and users get only the minimal permissions needed. An observer should not be able to escalate or move laterally. For instance, even after a valid login, if a user tries to perform an action outside their role, the system denies it (no standing access beyond what's explicitly granted) [2].

- **Continuous Monitoring:** Authorization is not a one-time check. Long-lived sessions or tokens should be short-lived or revalidated. The system continuously audits behavior; an account or service might need to re-authenticate or get re-evaluated when context changes (location, unusual pattern).

- **Micro-Segmentation:** Beyond software controls, deploy network policies that isolate IAMF components. For example, only the API gateway might be reachable from the public internet, and internal services only talk over isolated channels. Each component's interface is locked down by both network firewall and service-level access rules.

By strictly following Zero Trust principles, IAMF-Zero ensures that even if one component is compromised, it cannot easily impersonate others or move within the system unchecked. Every request is verified, and security is enforced at each layer [1] [2].

---

1 Zero Trust Cybersecurity: ‘Never Trust, Always Verify’ | NIST
https://www.nist.gov/blogs/taking-measure/zero-trust-cybersecurity-never-trust-always-verify

2 4 Zero trust, mTLS, and the service mesh explained
https://www.buoyant.io/blog/zero-trust-mtls-and-the-service-mesh-explained

3 RabbitMQ vs Kafka - Difference Between Message Queue Systems - AWS
https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-kafka/

5 6 10 How to Secure Communication Between Microservices - Styra
https://www.styra.com/blog/how-to-secure-communication-between-microservices/

7 JWT Authentication and Securing Microservices Tutorial | Kong HQ | Kong Inc.
https://konghq.com/blog/engineering/jwt-kong-gateway

8 9 ABAC vs. RBAC: What’s the difference? - Citrix Blogs
https://www.citrix.com/blogs/2022/05/17/abac-vs-rbac-comparison/?srsltid=AfmBOoq2OkK53suSvTfPivmIL7b5hWmKdFfuokWhXrCS9iuE67DmZLZa

11 What does API gateway do in Microservices Architecture? | by Mahesh Saini | Medium
https://medium.com/@maheshsaini.sec/what-does-api-gateway-do-in-microservices-architecture-d1e93e27e040

12 Security concerns while building Microservices - DEV Community
https://dev.to/navis/security-concerns-while-building-microservices-i62